Content view

The ViewController

eZ Publish comes with a native controller to display your content, known as the **`ViewController`**. It is called each time you try to reach a content from its **Url Alias** (*good looking* URI generated for any content) and is able to render any content previously edited in the admin interface or via the eZ Publish Public API.

It can also be called directly by its direct URI : `/content/location/<locationId>`

A content can also have different **view types** (full page, abstract in a list, block in a landing page...). By default the view type is **full** (for full page), but it can be anything (*line*, *block*...).

> **Important note regarding visibility**
> Location visibility flag, which you can change with hide/unhide in admin, is not permission based and thus acts as a simple potential filter. **It is not meant to restrict access to content**.
>
> If you need to restrict access to a given content, use **Sections or Object states**, which are permission based.

View selection

To display a content, the ViewController uses a view manager which selects the appropriate template depending on matching rules.

> For more information about the **view provider configuration**, please refer to the dedicated page.

> You can also use your own custom controller to render a content/location.

Location view template

A content view template is like any other template, with several specific aspects.

Available variables

Variable name	Type	Description
`location`	eZ\Publish\Core\Repository\Values\Content\Location	The location object. Contains meta information on the content (ContentInfo) (only when accessing a location)

content	eZ\Publish\Core\Repository\Values\Content\Content	The content object, containing all fields and version information (VersionInfo)
noLayout	Boolean	If true, indicates if the content/location is to be displayed without any pagelayout (i.e. AJAX, sub-requests...). It's generally `false` when displaying a content in view type **full**.
viewBaseLayout	String	The base layout template to use when the view is requested to be generated outside of the pagelayout (when `noLayout` is true).

Template inheritance

Like any template, a content view template can use template inheritance. However keep in mind that your content can be also requested via sub-requests (see below how to render embedded content objects). In this case your template should probably not extend your main layout.

In this regard, it is recommended to use inheritance this way:

```
{% extends noLayout ? viewbaseLayout : "AcmeDemoBundle::pagelayout.html.twig" %}

{% block content %}
...
{% endblock %}
```

Rendering content's fields

As stated above, a view template receives the requested Content object, holding all fields.

In order to display the fields' value the way you want, you can either manipulate the Field Value object itself or use a template.

Getting raw Field value

Having access to the Content object in the template, you can use its public methods to access to all the information you need. You can also use ez_field_value helper to get the Field value in the current language if translation is available.

```
{# With the following, myFieldValue will be in the content's main language, regardless
the current language #}
{% set myFieldValue = content.getFieldValue( 'some_field_identifier' ) %}

{# Here myTranslatedFieldValue will be in the current language if a translation is
available. If not, the content's main language will be used #}
{% set myTranslatedFieldValue = ez_field_value( content, 'some_field_identifier' ) %}
```

Using the FieldType's template block

All built-in FieldTypes come with a piece of Twig template code you can take advantage of by calling `ez_render_field()` helper.

```
{{ ez_render_field( content, 'some_field_identifier' ) }}
```

Refer to `ez_render_field()` reference page for further information.

> As this makes use of reusable templates, **using `ez_render_field()` is the recommended way and is to be considered as a best practice**.

Rendering Content name

The **name** of a content is its generic "title", generated by the repository considering several rules in the FieldDefinition. It usually consists in the normalized value of the first field.

There are 2 different ways to access to this special property:

- Through the name property of ContentInfo (not translated).
- Through VersionInfo with the TranslationHelper (translated).

Name property in ContentInfo

This property is the actual content name, but **in main language only** (so it is not translated).

```
<h2>Content name: {{ content.contentInfo.name }}</h2>
```

```
$contentName = $content->contentInfo->name;
```

Translated name

> The TranslationHelper service is available as of version 5.2 / 2013.09

The *translated name* is held in `VersionInfo` object, in the names property which consists of hash indexed by locale. You can easily retrieve it in the right language via the `TranslationHelper` service.

```
<h2>Translated content name: {{ ez_content_name( content ) }}</h2>
<h3>Also works from ContentInfo : {{ ez_content_name( content.contentInfo ) }}</h3>
```

You can refer to ez_content_name() reference page for further information.

```
// Assuming we're in a controller action
$translationHelper = $this->get( 'ezpublish.translation_helper' );

// From Content
$translatedContentName = $translationHelper->getTranslatedContentName( $content );
// From ContentInfo
$translatedContentName = $translationHelper->getTranslatedContentNameByContentInfo(
$contentInfo );
```

> The helper will respect the prioritized languages.
>
> If there is no translation for your prioritized languages, the helper will always return the name in the main language.

You can also **force a locale** in a 2nd argument:

```
{# Force fre-FR locale. #}
<h2>{{ ez_content_name( content, 'fre-FR' ) }}</h2>
```

```
// Assuming we're in a controller action
$translatedContentName = $this->get( 'ezpublish.translation_helper'
)->getTranslatedName( $content, 'fre-FR' );
```

Exposing additional variables

It is possible to expose additional variables in a content view template. See parameters injection in content views or use your own custom controller to render a content/location.

Making links to other locations

Linking to other locations is fairly easy and is done with native `path()` Twig helper (or `url()` if you want to generate absolute URLs). You just have to pass it the Location object and `path()` will generate the URLAlias for you.

```
{# Assuming "location" variable is a valid
eZ\Publish\API\Repository\Values\Content\Location object #}
<a href="{{ path( location ) }}">Some link to a location</a>
```

If you don't have the Location object, but only its ID, you can generate the URLAlias the following way:

```
<a href="{{ path( "ez_urlalias", {"locationId": 123} ) }}">Some link to a location,
with its Id only</a>
```

Under the hood
In the backend, `path()` uses the Router to generate links.

This makes also easy to generate links from PHP, via the `router` service.

Render embedded content objects

Rendering an embedded content from a Twig template is pretty straight forward as you just need to **do a subrequest with `ez_content` controller**.

Using `ez_content` controller

This controller is exactly the same as the ViewController presented above and has 2 main actions:

- **`viewLocation`** to render a location (same as when accessing a content through an URLAlias)
- **`viewContent`** to render a content

You can use this controller from templates with the following syntax:

eZ Publish 5.1+ / Symfony 2.2+

```
{{ render( controller( "ez_content:viewLocation", {"locationId": 123, "viewType":
"line"} ) ) }}
```

The example above allows you to render a Location which ID is **123**, with the view type **line**.

> Reference of `ez_content` controller follow the syntax of *controllers as a service*, as explained in Symfony documentation.

Available arguments

As any controller, you can pass arguments to `ez_content:viewLocation` or `ez_content:viewContent` to fit your needs.

Name	Description	Type	Default value
`locationId`	Id of the location you want to render. **Only for `ez_content:viewLocation`**	integer	N/A
`contentId`	Id of the content you want to render. **Only for `ez_content:viewContent`**	integer	N/A
`viewType`	The view type you want to render your content/location in. Will be used by the ViewManager to select corresponding template, according to defined rules. Example: full, line, my_custom_view, ...	string	full
`layout`	Indicates if the sub-view needs to use the main layout (see available variables in a view template)	boolean	false
`params`	Hash of variables you want to inject to sub-template, key being the exposed variable name. > Available as of eZ Publish 5.1 ``` {{ render(controller("ez_content:viewLocation", { "locationId": 123, "viewType": "line", "params": { "some_variable": "some_value" } })) }} ```	hash	empty hash

ESI

Just as for regular Symfony controllers, you can take advantage of ESI and use different cache levels:

Using ESI (eZ Publish 5.1+ / Symfony 2.2+)

```
{{ render_esi( controller( "ez_content:viewLocation", {"locationId": 123, "viewMode": "line"} ) ) }}
```

Asynchronous rendering

Symfony also supports asynchronous content rendering with the help of hinclude.js library.

Asynchronous rendering (eZ Publish 5.1+ / Symfony 2.2+)

```
{{ render_hinclude( controller( "ez_content:viewLocation", {"locationId": 123,
"viewMode": "line"} ) ) }}
```

hinclude.js needs to be properly included in your layout to work.

Please refer to Symfony documentation for all available options.